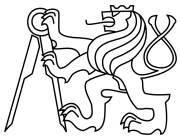
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Test framework for cloud failover mechanism
Student:	Štěpán Vaněček
Supervisor:	Dr.-Ing. Marius Feldmann
Study Programme:	Informatics
Study Branch:	Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2017/18

Instructions

The goal of the thesis is to design and develop a test framework evaluating the latency behavior of a chosen approach to failover of cloud resources provided within a distributed datacenter. The test framework will measure a temporal behavior of the failover regardless the selected approach. The measured values will be analysed and discussed.

The process consists of the following steps:
1. Analyse and describe the currently existing failover approach.
2. Analyse and design a universal test framework for the failover mechanism addressing the specific characteristics of a highly distributed datacenter running OpenStack.
3. Implement a software prototype using the infrastructure of Cloud & Heat Technologies GmbH.
4. Run the test framework on integrated failover mechanism and evaluate the latency measurements.
5. Analyse and discuss the measured values.

References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 2, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Test framework for cloud failover mechanisms

Štěpán Vaněček

Supervisor: Dr.-Ing. Marius Feldmann

17th May 2016

Acknowledgements

In the first place, I would like to thank my supervisor Dr.-Ing. Marius Feldmann for allowing me to write this thesis and for his guidance in the process of writing. Moreover, I would like to thank my family and my friends for their support along my studies and especially during the writing of this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 17th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Vaněček, Štěpán. *Test framework for cloud failover mechanisms.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Cílem této bakalářské práce je navrhnout univerzální testovací framework pro přístupy k failoveru cloudových služeb v rámci vysoce distribuovaného datového centra. Testy měří efektivitu provedeného failoveru z časového hlediska. Koncept je implementován jako softwarový prototyp, aby ověřil vhodnost návrhu. Framework je poté testován a výsledky testů jsou prezentovány. Testovací framework umožňuje otestovat různé přístupy k failoveru použité v infrastruktuře Cloud&Heat Technologies GmbH.

Klíčová slova Cloud computing, Failover, IaaS, VM, Vysoká dostupnost

Abstract

The goal of this Bachelor Thesis is to design a universal test framework solution for an ad-hoc failover of cloud services provided within a highly distributed datacenter. The framework measures the efficiency of performed failover out of a temporal perspective. The concept is implemented as a software prototype to verify the quality of the design. Finally, the framework is tested and the results are be presented. The test framework is able to test different failover approaches used within the infrastructure of Cloud&Heat Technologies GmbH.

Keywords Cloud computing, Failover, IaaS, VM, High availability

Contents

List of Figures

Chapter 1

Introduction

Cloud computing is a model of providing ad-hoc shared resources on demand over the internet. As well as while using any other technology, some unexpected behaviour may occur while using cloud services. The problems can cause a failure and thus unavailability of a part of the system, or even the whole solution. Cloud providers offer, beside other services, their infrastructure, thus the infrastructure is used by customers as a service. Customers usually use cloud infrastructure either for internal purposes or as a foundation for building up own products to offer. Such products include for example deploying a website or running a data storage application.

Although the customers of the cloud provider use the infrastructure in various different ways, it is desirable to build up a reliable solution for almost all of them. Regardless the purpose, it is always an advantage to have a reliable service providing the expenses are not taken in concern. Thus, it is crucial to know the temporal characteristics of the downtime in case of a failure of a node.

It is up to one's choice to decide whether to deploy a fault-tolerant solution or not. To avoid outages of solutions a customer has built up and uses or offers, he has to develop or integrate some mechanism handling these unexpected incidents. This policy keeps the system operating even if a node fails.

The failover policy helps to ensure high availability of the solution it is contained in. The overall goal of a failover mechanism is to transfer the services from a failing instance to a healthy one, so the system continues to be fully operational using a redundant instance instead of the failed one. This procedure is made with nearly no interruption in the services provided to users of the end solution.

1.1 Motivation

There is a lot of, both universal and cloud-specific, existing failover approaches. We can identify various different solutions and technologies. Each of the tech-

nologies has its advantages and disadvantages and it is thus quite uneasy to select the best fit for a particular service. The efficiency of the selected solution varies in regards with the infrastructure it is being installed on, the main purpose of using the service, the availability/costs balance preference, and a lot of other aspects.

Customers deploying their solutions to the infrastructure of Cloud & Heat Technologies GmbH (Cloud & Heat) have to decide by themselves which approach would be the best fit for a specific solution. Currently there is no reliable tool that could help them with making the decision. On this basis, a framework able to evaluate temporal characteristics of various failover approaches was developed. The framework is able to evaluate the failover approach on an exact solution of one's choice.

Such a framework is a tool for comparing multiple approaches used for performing the failover of various solutions. The comparison focuses on temporal information regarding the downtime of the service as a whole while switching the primary and redundant resources.

1.2 Goals

The goal of this Bachelor Thesis is to design and develop a test framework for cloud failover that fits the infrastructure of Cloud & Heat and to present and discuss the data resulting performed tests. Cloud & Heat Technologies GmbH is a cloud provider running a highly distributed datacenter with various deployments across Germany providing Infrastructure as a Service (IaaS) running OpenStack.

Additionally, there were test cases performed on the developed framework. The tests are able to simulate common use cases such as an outage of a node or a deployment running a web-server. The implemented solution was tested to get realistic data of the failover behaviour in case of an outage. This data was analysed to evaluate the tested approaches.

1.3 Structure of the thesis

The second chapter of the thesis defines the terminology that is used to specify different types of infrastructure and failover approaches. Then, an overview of existing approaches is presented.

The third chapter presents a typical use case that defines several demands for the framework. On the basis of the needs for the application, requirement analysis is made later in this chapter.

The fourth chapter gives a short overview of similar approaches. As no approach with similar aim is available, the overview introduces some research and publications made in the field of high availability.

On the basis of the requirement analysis and the presented failover approaches, the design of the solution is described in the fifth chapter.

The techniques of implementing the design with help of chosen technologies is described in the sixth chapter.

The seventh chapter describes evaluation of the framework and explains how the requirements were fulfilled. Moreover, it presents results of several measurements made with the framework.

Chapter 2

State-of-the-art overview of failover approaches

This chapter introduces some of the common failover mechanisms applicable to typical cloud infrastructures.

2.1 Overview of terminology

Failover is a process of switching the primary and the secondary component of a highly available service. In cloud environments, it could be referred to virtual machines (VMs) or to containers.

The terminology used for describing the failover approaches is explained in this section.

2.1.1 Level of automation

The execution of the failover can be implemented at a different level of autonomy. There are failover solutions triggered with assistance of a system administrator or after his confirmation. Such solutions are dependent on human interaction and thus are not autonomous. This is desirable for some systems, other systems can run autonomously. Downtimes of failovers requiring human interaction are generally significantly longer then the autonomous ones. This is caused by waiting for the administrator's interaction.

Failovers requiring human interaction are not a usual solution in the cloud environment. Thus, only autonomous failovers are discussed in this chapter.

2.1.2 Stateful and stateless service

There are two types of services addressed by failover mechanisms - stateful and stateless services. The type of service influences the technologies that may

be used so it is important to know the type of the service for implementing the suitable failover approach.

2.1.2.1 Stateful service

A service is considered stateful if there is a preceding action affecting the execution of current action. Stateful service is a state finite machine where every request leads the process to a new state. Considering this setting, a stateful service is possible to process two exact requests differently because of different states in which the services have initially been. When designing a failover solution for a stateful service, the architecture must meet the requirements. The mechanism must ensure that the state of the secondary instance is the same as the state of the primary instance. Only by ensuring this setting, the failover can be provided properly. [1]

2.1.2.2 Stateless service

Unlike stateful services, stateless services do not need to store any information regarding the current state of the service. All nodes are in the same state that does not change over time. That means that each request is treated as an independent one. In conclusion, each node is capable of handling any request. Stateless services are often created as highly available by load-balancing multiple nodes since there is no need to synchronise them. [1]

2.1.3 Service configuration

We can divide the failover approaches into two groups – active/active and active/passive – on the basis of its configuration. When differentiating these two groups "active" stands for a node that is online and operating. The "passive" node is a node that has to be brought online in case of a failure of the active node.

The passive node is often a node that is not operating and is in a stand-by mode. Such a node can be brought online by activating or rebooting it. It is also often a node that has an IP address that is not used by the service so no requests get redirected to that node. In that case, a node is brought online by reassigning an IP address or updating other network settings made on other service components. In this case, the node is active as soon as it has an IP address to which the communication takes place.

2.1.3.1 Active/passive configuration

The active/passive installation consists of an active primary instance and at least one passive redundant instance. In a normal state, the primary machine is in charge. The other machines are in a stand-by mode waiting for the activating signal. Should the primary machine be found unhealthy, a redundant

node has to take over the control. It is however not available immediately so it needs to be brought online. [1]

2.1.3.2 Active/active configuration

The active/active configuration consists of at least two nodes in the active mode. In this configuration, all nodes are immediately available and ready to take over. Since all the nodes are immediately available, the overall downtime during the failover tends to be shorter then the active/passive installation failover. Therefore, it is often desired to build up active/active solutions if possible. The active/active configuration is often set up as load-balanced.

A load-balanced configuration consists of a load balancer, usually a reverse proxy server, and balancing nodes. The load balancer distributes the workload to preferred nodes. It often includes a health check that periodically checks the health of all nodes so no request gets to a corrupted node. The nodes can be equal (thus, the load is distributed evenly) or primary and redundant. In the second case, the redundant nodes get no workload as long as the primary nodes are operating. [1]

2.1.4 Health checking

Health checking is a feature that enables to receive the status of a service. It is used for determining healthy and corrupted instances. The effort of the health check is to recognise and identify a failure of the node as soon as possible. The sooner a downtime is discovered, the sooner can the failover process be triggered and the shorter the downtime gets. The goal is that the outage is recognised soon enough so that the end user can experience no downtime of any service.

Once the health check reveals a failing or an already failed node, the failover process begins. The purpose of the process is to isolate the failed node and handle the further communication only with the healthy and operating nodes.

2.1.4.1 Heartbeat

The typical health check for failover clusters is heartbeat. It consists of sending requests to the previously defined nodes determining the status of the machine in a given moment. The heartbeats are sent with defined periods. In multi-node clusters, it is possible to design two heartbeat architectures:

1. A master node receives heartbeats from all nodes.

2. All nodes receive heartbeats of other nodes.

2.2 High-availability difficulties

When increasing the availability of a solution by implementing failover approaches to services of the system, there are several difficulties that the administrator has to take in account. Otherwise, the failover solution could end up with undefined behaviour.

2.2.1 Maintaining the data consistency

A service usually contains both variable and invariable data. The invariable data does not change over the time and it is thus easy to ensure the consistency of the data. Since the data does not change, it is enough to import the up-to-date data when creating the node as a part of a highly available installation to assure the consistency of the data.

The variable data usually include databases, volumes, or simply variables. These data change over time so even if all redundant nodes are launched with the same data, it has to be ensured that the data remain consistent for the time the system is operating. The data get created, updated, and deleted on one machine. Therefore, it is necessary to create or implement a mechanism that ensures the redistribution of the changes to all nodes handling the same service.

The easiest and the most common solution is to have one master node that is up-to-date and handles all the workload. The other nodes – the slave nodes – receive the information of what has changed in the master node's data structure and thus can be kept up-to-date as well. When a failure of the master node occurs, one of the slave nodes becomes the master node and the system continues operating.

2.2.2 Split brain syndrome

The split brain syndrome is a potential source of errors in failover clusters. It is often a result of incorrect communication of the cooperating nodes, for example a failed heartbeat communication path. The threat is illustrated in [2, p. 160-161]: "*Even if the intercluster communication path is redundant, it can fail – and then we are in deep trouble. A two-node cluster illustrates that best: each node will think that the other node is dead, and will start all services to make them available again. That behaviour is called a split brain situation. It means that the shared disk space will be accessed by both nodes, both will activate the same IP addresses on the same shared network, etc. As the minimum result, services will not work. But then you will be lucky – it may also be that persistent data is damaged beyond repair.*" This behaviour can be prevented by assuring that there is always only one working node in the cluster. Should the node fail, it has to be shut down and removed from the cluster so that it is unable to affect other infrastructure.

2.2.3 Switching the traffic

When a failover is being performed, one node is often being replaced with another. The services interacting with the affected service have to be prepared for this situation. That means that it has to be ensured that the interacting services will communicate with the redundant instance once it is made primary. This procedure may include updating the IP address where service is available. If not all communication is transferred to the new primary node, the system is not able to cooperate and thus does not work properly.

2.3 Existing failover approaches

We can divide the existing failover and high-availability approaches for VMs and containers by many factors. Although the stateful and stateless or active/active and active/passive division could be used for exact installations, most of the failover approaches enable to be used in multiple ways and thus in both configurations. The division used below takes the core logic of the approach into consideration.

There is a lot of different solutions providing failover at the moment. It is not a goal of this chapter to map and introduce all of them. The goal of this overview is to characterise various different approaches and to introduce some implementations fitting the specification.

2.3.1 Quorum clusters

This group of failover approaches is characterised by clustering primary and redundant nodes. The clusters are managed by the cluster manager and are able to communicate with one another. Either the cluster manager communicates with the nodes or the nodes communicate with each other. The communication usually includes heartbeat.

The quorum policy ensures correct behaviour of the nodes in the cluster. The quorum contains an odd number of nodes, starting at three, where the correct behaviour is defined as behaviour of the majority of the nodes. The odd number of the nodes ensures that there is no split possible. Once there is a different node discovered, it is left out of the cluster. There are various fencing agents enabling to achieve this. Usually, the node is removed from the cluster and isolated. The isolation lies in preventing the node changing any common data or other part of the solution. Otherwise, there would be a risk of corrupting the data in the application.

2.3.1.1 Corosync and Pacemaker

Pacemaker is an open-source cluster manager created by ClusterLabs. It is being distributed along with Corosync that is used for communication between

the nodes of the cluster. "*The ClusterLabs stack, incorporating Corosync and Pacemaker defines an Open Source, High Availability cluster offering suitable for both small and large deployments.*"[3]. Because of its simplicity and scalability, it became widely used.

Nodes of the cluster must be eligible to check the current status and to provide the failover itself upon the result of the status check. This behaviour is specified in resource agents. "*A resource agent is a standardised interface for a cluster resource. It translates a standard set of operations into steps specific to the resource or application, and interprets their results as success or failure.*"[4] The mentioned set of operations include operations starting and stopping an instance and determining the status. Pacemaker supports OCF resource agents, LSB resource agents and Heartbeat resource agents.

Corosync clusters use fencing agents. It defines whether the node is working properly and eventually isolates the corrupted one. STONITH fencing is used by default. This fencing agent monitors the consistency of the nodes. Should it find an inconsistent node, it is removed from the cluster and shut down so it cannot cause any damage to the rest of the system. An inconsistent node is defined as a node with different behaviour compared to the majority of nodes.

2.3.1.2 Zookeeper

Apache Zookeeper[1] is an open-source project aiming to enable highly reliable distributed solutions. It uses distributed processes that share information in a file system-like namespace. Because of this attribute, the number of nodes it is able to serve is limited. Nevertheless, the solution is very low-latency and fast.

Zookeeper provides failover by managing the primary and secondary resources in the cluster. If the primary instance fails, Zookeeper can be made responsible for choosing the new primary resource and announcing the change to other nodes so that they would communicate with the new primary node instead of the old one.

The target market for ZooKeeper are multi-host, multi-process C and Java based systems that operate in data centres.[5]

2.3.1.3 Consul

Consul[2] is a solution by HashiCorp providing high availability. It extends Serf (also by HashiCorp). The solution uses raft protocol in order to provide clustering and to ensure the quorum policy. Moreover, it uses the gossip protocol to manage the communication inside the cluster. The gossip protocol

[1] http://zookeeper.apache.org/
[2] https://www.consul.io

consists of two parts – one communicating over the LAN and the other one over the internet – and ensures the health checking.

2.3.1.4 Other approaches

There is a lot of other approaches that are widely used. The offered solutions differ in many aspects but share the common idea of clustering multiple nodes. The other solutions include:

- Doozerd[3],
- Etcd[4].

2.3.2 Load balancers

A load balancer is usually a proxy server standing between the source and the target node. Its task is to distribute requests from the source to the target node. For the client it appears like the request is handled by the proxy server itself, not by an upstream node. When sending the response, the upstream server responds only to the proxy server and it has to reroute the response back to the client node that awaits it.

The proxy server is used for distributing the workload between multiple service instances. The workload is, for the sake of load balancing, distributed equally if possible so all the nodes are equally stressed. In other cases, the role of primary and redundant services is distinguishable. In other words, it is usually easily possible to influence the stress put on each node.

2.3.2.1 HAProxy

HAProxy is a lightweight solution offering load-balancing. It is very fast but still able to serve numerous nodes. It is used as a reverse proxy server and thus redistributes the requests further to load-balanced upstream server instances.

HAProxy provides failover by distributing the workload across various instances and checking health of those instances. Once an instance is found unhealthy, the workload is no longer distributed to that instance until it is working again. Nevertheless, HAProxy alone does not provide high availability for the whole installation. Although the end service is highly available and a failure of an application instance does not affect the solution as a whole, all the communication must go through the HAProxy load balancer node. This makes HAProxy a single point of failure of the whole installation. In other words, if there is a downtime on the load balancing node, the whole service becomes unavailable.

[3] https://github.com/ha/doozerd
[4] https://coreos.com/etcd/

HAProxy is often used along with Corosync. In this set-up, it is easier to control and maintain the upstream nodes. Another use for Corosync in HAProxy installation is to ensure a failover instance for the HAProxy node itself. Since HAProxy is a single point of failure, it needs a redundant instance to fail over to when achieving a highly available installation.

2.3.2.2 Nginx

Nginx[5] is used as both an HTTP server and a reverse proxy server. For the sake of high availability, the reverse proxy functionality is used. The administrator is able to define the load-balancing algorithm and the upstream servers to connect to. Nginx redirects the requests to specified upstream servers. Load-balancing algorithms include round-robin, IP hash or least connection.[6, p. 67-84]

Nginx as a reverse proxy processes requests further to specified nodes. The responses are sent back through the proxy to the client node. This means that Nginx node is a single point of failure if there is no redundant proxy server node in the infrastructure ready to take over the workload. Still, Nginx provides an active/active failover for upstream server instances. Nginx checks the health of the servers and once an unhealthy server is found, no further request is redirected to that server.

2.3.2.3 Apache

Apache HTTP server can be modified to work as a reverse proxy server. That is achieved with the mod_proxy module. It is a module used for redirecting the connections. It can turn the Apache webserver into both forward and reverse proxy server. With the reverse proxy configuration, it can be used as a load balancer.

The reverse proxy configuration offers multiple load balancing algorithms. Available protocols for load balancing include HTTP, FTP and FCGI.

2.3.2.4 Zen

Zen load balancer[6] is an open-source solution for providing high availability via load balancing. It features the classical load balancing features, such as health checking. Compared to HAProxy and Nginx, it offers a GUI to ease the set-up.

It is distributed optimised for virtual infrastructure and thus cloud services, for being installed on physical hardware, or as a load-balancer-as-a-service.

[5] http://nginx.org
[6] https://www.zenloadbalancer.com

2.3.2.5 Other load balancers

There is a lot of other load balancer software. Although every solution has its specifics, its overall function is the same – providing the load balancing. Some of the other solutions enabling load balancing include:

- Caddy server[7],
- Squid[8],
- Pound[9],
- Lighttpd[10].

2.3.3 Routing protocols

The failover approaches stated below do not share any exact technology for providing the failover. Its common feature is that they are being performed on the basis of network protocols influencing routing.

2.3.3.1 BGP failover

Border Gateway Protocol is a protocol that is configured at the router. The BGP routing table contains neighbours defining the path where to route a packet. The content of the table of neighbours is usually inserted by an administrator so that he is able to operate with it. The BGP defines a node or a set of nodes to which to route the packets. The protocol routes the packets through the neighbouring node that is the most preferred.

BGP failover uses the characteristics of the BGP. It takes more ways how to access an IP address into account. When designing a BGP failover, the routing table must have multiple entries for one IP address. The most preferable path is addressing the primary node and the worse paths are addressing the redundant nodes. Should the primary node fail and thus the most preferable path become unavailable, BGP routes the packets towards the second best way and thus towards the redundant node. In this setting, the BGP provides an automatic fallback as well. As soon as the most preferable path becomes available again, the packets begin to be sent through the primary path and thus to the primary resource.

It is also possible to configure BGP failover as a load-balancing tool. The administrator must configure the metrics for the paths evenly in order to achieve that.

[7] https://caddyserver.com
[8] http://www.squid-cache.org
[9] http://www.apsis.ch/pound/
[10] https://www.lighttpd.net

2.3.3.2 OSPF failover

Open Shortest Path First failover is based on the OSPF dynamic routing protocol. The OSPF protocol routes the packets towards the easiest reachable node – it uses the shortest path according to the metric in the routing table at the router.

When having two nodes – the primary and the redundant one – the primary one has a better metric in the routing table so the OSPF protocol routes the packets to that node. When the primary node becomes unavailable, the routing table is updated. The metric changes because it is now impossible to reach the primary node. The redundant node is having worse metric when both nodes are operating but after the failure of the primary node, the metric of the redundant node becomes the best possible. Therefore, as soon as the table gets updated, the packets are routed to the redundant resource.

As well as the BGP failover, OSPF failover provides an automatic fallback. The primary resource begins having the shortest path again as soon as it is back online and accessible.

2.3.3.3 Other protocols used for failover

There are other protocols that have suitable characteristics to provide failover. Such protocols include:

- DHCP[11] – DHCP failover is used primarily in LANs.
- RIP[12].

2.3.4 DNS failover

DNS failover approaches provide the failover by changing the address of desired DNS hostname in the DNS table. Either there is a single IP address for a DNS hostname or there are more targets for one entry. If there are more targets for one hostname, some balancing algorithm such as round-robin will be most likely used.

Nodes are being monitored and once there is a downtime on the node that is kept in the DNS table, the record is removed, eventually replaced with a redundant instance.

DNS failover is however unreliable when discussing the temporal behaviour because of the distribution of the DNS table. Once a DNS failover is triggered, the DNS table gets updated. Still, records from the DNS table on client nodes are often being cached and refreshed first after a certain amount of time. The period of refreshing varies upon the set-up of the client node. This causes a delay in propagation of the updated DNS entry. Even if there is a DNS table

[11] Dynamic host configuration protocol
[12] Routing Information Protocol

up-to-date with current situation of the service, clients may be in hold of an obsolete version still leading to a failed service.

The DNS failover is successfully completed when the obsolete DNS record is no longer cached. This characteristic makes the temporal perspective of DNS failover varying for different client nodes.

Chapter 3

Use Case and Requirements analysis

The test framework was designed to evaluate the temporal behaviour of a chosen failover approach. It is intended to run and measure test cases, as well as to measure the temporal behaviour of failover on fully operating applications.

Below in this chapter, there is an outline of an example use case for the framework described. The described example defines requirements on the system. The requirements are specified in the second section of this chapter.

3.1 A company offering a highly available data storage application

A company uses Cloud & Heat's infrastructure to run their web application and to store data. The company runs a data storage-as-a-service and wants to measure the temporal behaviour of the failover in their system. By gathering temporal data for different failover approaches, the company is able to decide which approach to choose. Moreover, it can also predict a length of a downtime in the production deployment in case of a node failure.

The company has multiple key services that it wants to keep highly available – the data storage web application front-end and the database where all the data is stored. For this purpose, the company created test cases where the failover approaches can be tested.

3.1.1 Web application front-end

The 'web application front-end' test case deploys the web application front-end and the necessary back-end. The back-end is composed mainly of the data storage and an engine enabling to get relevant data for a user. The back-

end provides the web application front-end with relevant data enabling to browse one's file system. The application front-end interprets a web interface visualising the file system enabling to upload and download a selected file.

The web application runs on three instances in total in three different deployments. These instances are incorporated into a cluster and load balanced by a load balancer of the chosen failover software. The load balancer mechanism distributes the workload evenly between running instances and redirects the requests to chosen instance.

Once the temporal test is triggered, one of the instances stops responding to requests and communicating over the network. At this point, some requests (the amount is dependent on the type and the configuration of the load balancer) would be redirected to a service that is not communicating and thus is not executing the requests as it is supposed to. The health check of the failover approach has to detect the outage as soon as possible for the failover approach to receive the best temporal results possible. Once the health check considers the service unhealthy, it is removed from the cluster and the load balanced service continues operating only with healthy nodes in it. At this point, each request is handled correctly again until a new problem occurs. With regard to chosen failover approach, the health check might restart the service as an attempt to fix it. Should the restart solve the problem and the instance is handling requests correctly again, it is incorporated back into the cluster and continues handling requests.

3.1.2 Shared database

The data regarding the user accounts is stored in a highly available database cluster. The 'shared database' test case simulates a downtime of a database server. Once there is a create, update or delete request sent to primary instance of the shared DB, the information has to be distributed to redundant instances as well. Such a mechanism ensures that all DBs of the cluster have the same data and thus are up-to-date. The test case deploys two database instances - the primary and the secondary one. When operating, the primary DB handles all the requests. After a failure of the primary node occurs, the secondary instance takes over the control.

Once the temporal measurement is triggered, the primary instance is sent to a stand-by mode and does not communicate over the network. The web application instances send CRUD requests to the primary instance but it has stopped to communicate. The cluster gets aware of this change and a selected failover mechanism transfers the communication to a secondary instance. Should there be a time gap between the simulated failure of the primary database node and the point when the secondary instance of the cluster begins to handle the requests, the gap marks the downtime while performing the failover.

3.2 Requirements analysis

The prior section presents a use case that indirectly shows and defines some needs for the implementation of the framework. The needs are analysed and summarised into requirements that are introduced in this section.

The use case describes a flow of a temporal test measuring the duration of a downtime of selected failover approach in defined installation. The primary demand for the framework is to enable measuring the duration of the downtime. The results of the test must be presented to the administrator so that he can evaluate whether the failover approach he has chosen and implemented fulfils his expectations. In addition, the administrator wants to compare several failover mechanisms applied on the same service to decide which one is the best fit. On this basis, the outcome of the measurement has a unified format so that it is easy to compare several results. It is desirable for the result to be easy to read and to understand. On the other hand, it has to be able to be processed further as well, for example for the purpose of visualising the results.

Although it is not required, it is expected that the framework will evaluate mostly failovers over the internet. That means that the primary and the secondary nodes of a highly available installation are located in different deployments and thus are not in the same LAN. In such a scenario, the failover causes moving the service to another geographical location. The location of the nodes may be a factor influencing the duration of the downtime during the failover as well. In order to be able to find and eliminate potential deviations, the location of the affected nodes is required for evaluating the temporal results as well.

The company described in the use case has its service deployed in the infrastructure of Cloud & Heat. The company wants to run the measurement in the same infrastructure as the one the production application is running in normally.

The development team members work mostly on common Linux distributions and a minority uses Apple computers. Regardless the operating system, everybody would appreciate the possibility to run a measurement from his preferred platform.

The company offers a storage-as-a-service and have been receiving hundreds of requests per second so even a small difference of a downtime on their primary service is important and can affect the final decision for a solution to be chosen. It is thus desirable to perform the measurement with high precision.

3.2.1 Functional requirements

Req. number	Name
F1	Measure the duration of a downtime.
F2	Store the information regarding the physical location of all interacting instances.
F3	Store the information regarding the accuracy of the measurement.
F4	Provide the results in a unified format.
F5	Run the measurement for different configurations of failover approaches.
F6	Do not affect other services running in the same infrastructure.
F7	Release all allocated resources after the test.
F8	Store the configuration of the measurement.

3.2.2 Non-functional requirements

Req. number	Name
N1	Runs on a typical cloud platform.
N2	Communicates over the internet.
N3	Can be launched from Linux distributions and from OS X.
N4	The test is performed with a low margin of error.

3.2.3 Description of the requirements

F1 Measure the duration of a downtime.

The framework measures the time when the selected service is unavailable.

F2 Store the information regarding the physical location of all interacting instances.

The result of the test contains the former location of tested instance and the new location so it can be reflected while evaluating the results.

F3 Store the information regarding the accuracy of the measurement.

The result of the test contains the actual accuracy of the measurement and the measured value.

F4 Provide the results in a unified format.

The result is kept in a unified data format. The format enables easy further manipulation with the data included in the report of the performed test.

F5 Run the measurement for different configurations of failover approaches.

The framework is able to evaluate the temporal behaviour of both active/active and active/passive failover configuration installed on the typical cloud infrastructure.

F6 Do not affect other services running in the same infrastructure.

While performing a temporal test, no other services or applications are affected by the test. The measurement runs isolated from other services running in the cloud.

F7 Release all allocated resources after the test.

The framework ensures leaving no data or applications in the infrastructure after a measurement is completed. It deletes all VMs and resources allocated for it.

F8 Store the configuration of the measurement.

The data of the measurement is stored along with the result. Beside other data specified before, it contains the set-up of the VMs and notes taken to describe the performed test.

N1 Runs on a typical cloud platform.

The tests using the framework are able to be triggered in the infrastructure of Cloud & Heat. Such an infrastructure is built on OpenStack.

N2 Communicates over the internet.

The framework communicates over the internet. Presence of all involved machines in the same LAN or establishing a VPN connection is not required for the measurement. Moreover, it is even not desirable because mainly the failover in the internet is expected to be tested.

N3 Can be launched from Linux distributions and from Mac OS X.

The test is possible to be triggered from Mac OS X El Capitan and from Ubuntu Linux 14.04.

N4 The test is performed with a low margin of error.

The framework measures the duration of the downtime with milliseconds accuracy.

Chapter 4

Existing solutions

At this moment, there is no free tool enabling evaluation of temporal aspects of failover approaches available.

However, there are existing monitoring tools on the market. The monitoring tools enable to measure service uptime and downtime during its operation. It is often used in highly available installations in order to provide information about the status of both the service itself and its particular components.

The known server monitoring tools include Nagios, Icinga and Monitis.

On the other hand, it is not really a suitable solution for performing the tests. When considering the requirements from the previous chapter, the monitoring tools do not meet all of them. The most important difference is that the monitoring tools are designed to monitor. Thus it monitors running infrastructures and is not designed to build testing infrastructures. Therefore the rest of work is left to the programmer.

Nevertheless, a monitoring tool is a very useful thing when maintaining a cloud solution.

Additionally, there is no research covering the temporal behaviour of different failover approaches at the moment. There was however some research made in the field of availability of services. According to [7], there is a classification of a specified service availability. We can differentiate 7 classes of service availability by determining its total availability during the year. The availability classes are defined in the following table.

Availability class	Availability [%]	Unavailability [mins/year]
1	90	50000
2	99	5000
3	99.9	500
4	99.99	50
5	99.999	5
6	99.9999	0.5
7	99.99999	0.05

Different solutions that require different level of availability. The services may be designed to fit in one of the classes of availability. Higher classes of availability already present a low unavailability duration. For example, availability class 7 requires less than 4 seconds downtime per year.

On the basis of the absolute service downtime length over a longer period, the probability of its availability can be predicted.

A research published in [8] examines clients' experience with downtime of major cloud providers' services in years 2007 - 2012. The experience is based on the overall availability of the service during the year. This data represent all layers of cloud architecture – IaaS, PaaS and SaaS. However, this data can not serve to determine the downtime during the failover because there is a lot of different services included in the analysis. It is then not possible to extract values regarding particular service.

Chapter 5

Design

On the basis of the functional and non-functional requirements stated in chapter 3 and the state-of-the-art failover overview in chapter 2, the framework was designed. It aims to construct a universal, light-weight, and efficient system that is easy to work with.

The framework aims to provide a measuring tool for systems running in the cloud. It evaluates the temporal behaviour of failover triggered by a downtime of a chosen system's node.

The system must contain a highly available installation on which the temporal behaviour is tested. It is also possible to measure multiple failovers on different parts of the system, though it has to be done by performing multiple tests.

The main focus of the test is to measure the time during which the system is unavailable. Such a metric can be used in order to predict the length of the downtime in case of a production node failure.

The independent and reliable result is achieved by deploying a new instance of the system to be tested for every measurement so that the results do not get distorted and the measurement always tests a fully operational system.

5.1 Duties of the framework

The measurement itself expects a fully operational system at the beginning. Therefore, it is responsible for building up the whole solution as the specified use case requires. While building up the system, the framework stores information regarding the VMs launched for further use in later phases.

The measurement is adjusted to await and measure the duration of the downtime. To ensure that the failover occurs, it is the application's respons-

ibility to trigger it. The failure is simulated on a VM preferred in the setting of the use case.

As the framework takes control of building up the solution where the downtime length is measured, it has to ensure its full removal after the measurement. For this purpose, the information regarding the launched VMs is used. Every VM that has been created is removed at the end of the process.

5.2 Supervision of the availability

The measurement is controlled from outside the measured environment. That means that the decision regarding the availability is made on a client node. This attribute provides the measurement with the following characteristics:

- The framework is universal.
- The measurement has lower precision.

The framework checks the availability of the desired highly available system. In theory, there is no need to know any details regarding the failover approach that is being used or the infrastructure bearing the solution. Nevertheless, the administrator has to specify some attributes of the test case in order to facilitate terminating the measurement after the failover process is completed.

The reduced need for details of the failover approach is caused by the black-box scheme of the testing. Since the measurement is controlled from the outside of the environment, the framework has no control over the processes happening inside. It targets a joint feature of servers – the ability of responding to requests. The framework defines and integrates its own response structure.

Because the framework uses the internet for communication, the measured values contain all inaccuracies caused by the transport over the network.

The controlled subject – the highly available installation – is observed by the framework in order to receive the data regarding the availability of the solution. The framework is able to evaluate availability of the tested system upon the results of the communication itself.

This approach brings a slight loss of precision because the framework and the controlled subject are connected over the internet. It is unable to avoid a delay when communicating over a network, especially the internet. The delay has varying temporal characteristic so that its impact cannot be entirely eliminated. Moreover, the framework gets only the data considering the availability of a server in a certain moment - in other words, it does not recognise the exact moment of the server falling down, it only knows the last time the server was up and thus responded to the request and the time of the first request to which the server responded again.

5.2.1 Other supervision possibilities

In contrast to the black-box testing, the other approach to determine the availability and measure the unavailability is to collect the data right at the source – from inside the system. In such a case, however, it is already desirable to gather information directly from the components responsible for providing the high availability. Otherwise, it would introduce no improvement in comparison with the chosen solution.

Provided the measurement is controlled from inside the installation – directly at the component determining the target of requests, it is able to get more accurate information taking all nodes of the highly available installation in consideration. The high precision is an advantage of the presented approach.

However, it still remains difficult to get exact temporal data of two important moments during the failover process - the beginning of the downtime and the end of the period when the system is unavailable, eventually only partially available. The reason of such an intricacy is that when a server is failing, it is not possible to let it send a message about its state going down. Either the message is sent and it means that the server might still beat least partially operating or the server might be down and that is why the message could not be delivered. Such a complication leads the design back towards the distant availability checking as used in the concept. The only difference is that the instances within the system check one another from inside. Nevertheless, the final precision is higher because the involved nodes can be targeted directly and the communication does not necessarily take place on the internet.

The disadvantage of the presented solution is the loss of universality. The failover approach dependency happened to be the reason why not to take the measurements from inside the box in concern for the framework.

To use such an approach, every installation intended to be tested would have to be specified within the framework so that it has an overview of the system and components it uses for providing the failover. Only then it is possible to know the exact nodes it should focus on and the behaviour it should await. Moreover, there is a lot of failover approaches on the market nowadays. Different failover approaches use different techniques to monitor nodes, to provide the failover, or for example to evacuate corrupted nodes from the cluster or to recover failing servers. These differences in the design of the solution would affect the behaviour of the failover measurement so that different approaches require to check availability of different nodes and situations. Finally, when adding various checks on particular nodes of the infrastructure, there would have to be high interaction of the administrator who triggers the test and the framework mastering it.

At the current state, it is not necessary to map all the solutions on the market and take all of its settings in concern so that a universal framework could be built up. In addition, such a solution would get obsolete as soon as

new technologies are available.

5.3 Logic of the measurement

As mentioned in the subsection 5.2, the measurement itself consists of sending requests to verify the availability of the service. A single request is able to determine whether the system is operating at a certain moment. In total, more requests provide the measurement with more records on the time axis where every entry carries the information of system's availability.

While aiming to have a high precision of the measurement, it was necessary to enable as high frequency of sending the requests as possible. On the other hand, if too many requests are sent to the target service, an upstream VM or any other component can get overloaded. As a result, the frequency has to be set with caution.

Because the framework checks the availability of the system and triggers the failover, it is desirable to find out when the failover process is completed. That defines another task - checking the data received from the requests and determining the completed failover process on its basis. The algorithm needs to be able to discover the completion of all usual failover approaches. It is however not necessary to discontinue the process of sending requests immediately when the failover process is complete. Terminating the measurement with delay only brings redundant data to the results structure. In fact, it causes no harm to either part of the installation.

As a result of the need of sending the request as close from one another from the temporal perspective, the mechanism handling sending the request needs to trigger the sending with regular time intervals. The precision of the measurement is dependent on the length of the gap between successful and unsuccessful response from a VM. That is why the requests-sending logic runs separated from the logic determining the finished failover. Otherwise, there would have been an interruption in sending the requests every time the state of the measurement would have been evaluated. The prevention of interference between the logics running simultaneously is solved by running both algorithms in separated threads.

5.3.1 Storing the data

There are multiple threads running while the measurement is being performed. One thread takes care of periodically sending requests and receiving responses from the highly available installation. The other thread evaluates the responses in order to establish the end of the process and to count the desired values. That is described in subsection 5.3.2.

The first thread needs to have a writing access to the data in order to write new entries it gets as a result of sending the requests. Moreover, the second thread needs to be able to read those data at the same time. For this

purpose, the data is stored in the memory of the program. That ensures an easy access to the data structure for both mentioned processes.

5.3.2 End of the measurement

When the requests have begun to be received, the data begin to be stored and analysed.

While the system is still fully operating, the framework must gather enough data in order to be able to determine later which type of failover approach it is measuring. It is important in order to be able to diagnose the completed process of failing over and thus to stop the measurement once there is nothing to measure. There are several eventualities that may result in different scenario or responding. Because the framework has no information about the tested approach, it has to decide on the basis of the scenario of responding.

5.3.2.1 One primary instance and a period with no response detected

The scenario with one primary instance[13] is the most common and the simplest case.

A system is responding positively to requests before the failover mechanism is triggered. Then, at least one request receives no response. This point marks the beginning of the downtime and thus the time important for the measurement.

The system is back online again as soon as a response to a request from another machine is received. Still, multiple positive responses are required to terminate the test. After enough positive requests are received, the measurement can be stopped.

5.3.2.2 Multiple primary instances and a period with no response detected

In this scenario, there are responses from more machines received before triggering the failover. Because more primary machines are involved, the workload is most likely distributed between these nodes. When the failover is triggered, one node is put into downtime but the others are most likely still in operation.

The start of the failover is the same as in subsection 5.3.2.1 – a request that received no response.

However, the end of the failover has to be followed by a defined number of positive responses. The number is calculated reflecting the frequency of the responses from the node set to downtime in overall positive responses before

[13] A primary instance is an instance that receives requests before triggering the downtime.

triggering the failover. The following equation defines the necessary amount of positive responses:

$$N = \frac{s_1 + s_2 + ... + s_k + ... + s_n}{s_k} * 3, \tag{5.1}$$

s_i is number of requests received from given service in the measured period
s_k is the service in downtime

The equation calculates the probability of targeting the request to the machine in downtime. Receiving this number of positive responses ensures that the corrupted node is already put away from the cluster. Otherwise, there would be 2 or 3 requests not responded because the unhealthy node was addressed.

Again, as in subsection 5.3.2.1, the end of the downtime is set down to the timestamp of the request that would normally be unsuccessful but now is responded normally. Such a request is K requests after the last unsuccessful response, where $K = N/3$.

5.3.2.3 Incorrect responses detected

This scenario most likely implies a situation where the node addressed by the request is not a node in the tested highly available installation or it is a node that has been kept in a stand-by mode and is only partly operating.

In either case, the incorrect response marks the beginning of the downtime in similar way as in subsection 5.3.2.1 – a response that is not correct according to the defined template marks the beginning of the measured period.

The end of the downtime is calculated in the same way as in subsection 5.3.2.2. Only in this case an incorrect response is marked as unsuccessful as well.

To mark the safe moment for ending of the measurement, the equation 5.1 is used to calculate enough number of positive responses to the requests received in a row. In this case, s_k is the service responding incorrectly.

5.3.2.4 No incorrect response received

When no incorrect response is received during the measurement, no downtime has been detected. Such a behaviour indicates one of the following scenarios:

1. The failover was performed with no downtime at all.

2. The downtime occurred for a very short period of time.

   The length of the downtime was shorter than the precision of the measurement.

Regardless the scenario, the result of the measurement is a zero downtime. As to all other results, the precision is added to define the value of the result.

Even when no unexpected responses occur, the framework must be able to determine the point when the failover have been performed. The completed failover is acknowledged at the moment when a response from any of the primary machines is not contained in N last responses where:

$$N = \frac{s_1 + s_2 + ... + s_k + ... + s_n}{s_k} * 3, \tag{5.2}$$

s_i is number of requests received from given service in the measured period
s_k is the service being inspected

The number is calculated in the same way as in 5.1 but it has to be calculated for every server so that presence of all primary servers in the cluster can be verified.

5.4 Calculation of results

5.4.1 Downtime length

The overall downtime length is calculated on the basis of the results received from the measurement. It is set as the time difference between the first invalid response and the timestamp of a valid response at the point where normally an invalid one would be received so the system began to process all the test requests properly again.

5.4.2 Precision

Precision is calculated differently on the basis of the scheme defined in subsection 5.3.2.

In scheme 5.3.2.1, the precision is set as the sum of the time difference between the last successful response before the downtime and the first unsuccessful request and the last unsuccessful and first successful request.

$$P = \frac{(t_2 - t_1) + (t_4 - t_3)}{4}, \tag{5.3}$$

t_1 is the timestamp of the last successful request before downtime,
t_2 is the timestamp of the first unsuccessful request,
t_3 is the timestamp of the last unsuccessful request,
t_4 is the timestamp of the first successful request after the downtime.

The schemes described in subsections 5.3.2.2 and 5.3.2.3 are calculated according to the following equation:

$$P = \frac{((t_2 - t_1) + (t_4 - t_3)) * \frac{s}{s_k}}{4}, \tag{5.4}$$

t_1 is the timestamp of the last successful request,
t_2 is the timestamp of the first unsuccessful request,
t_3 is the timestamp of the last unsuccessful request,
t_4 is the timestamp of the request following the last unsuccessful request,
s is the number of requests prior triggering the downtime,
s_k is the number of requests triggered prior the downtime responded by the failing node.

The precision of tests corresponding with scheme 5.3.2.4 is calculated in the following way:

$$P = \frac{(t_2 - t_1) * \frac{s}{s_k}}{2}, \tag{5.5}$$

t_1 is the timestamp of the last successful request from the failing node before downtime,
t_2 is the timestamp of the request following the last successful request of the failing instance,
s is the number of requests prior triggering the downtime,
s_k is the number of requests triggered prior the downtime responded by the failing node.

5.5 Workflow of the test

5.5.1 Workflow overview

When the test framework is triggered to measure the temporal behaviour of the selected failover mechanism, the highly available system for measuring the failover has to be built up. This consists of launching and setting up new VMs to demonstrate the highly available installation. All specifications for performing the test are stated in a configuration file. This enables an easy and fast manipulation with the parameters of the test.

The measurement itself is started on the instances creating the highly available installation. The measurement checks the availability of the service and measures the time during the system's unavailability. This is achieved by sending requests repeatedly to the service. The state of the service is determined on the basis of received responses.

After the measurement, all VMs are deleted so no useless data remains in the environment. The measured data is analysed and summarised.

After the test is finished, the output file is created and all required data is filled in. Finally, the result – the output file – is distributed to the administrator.

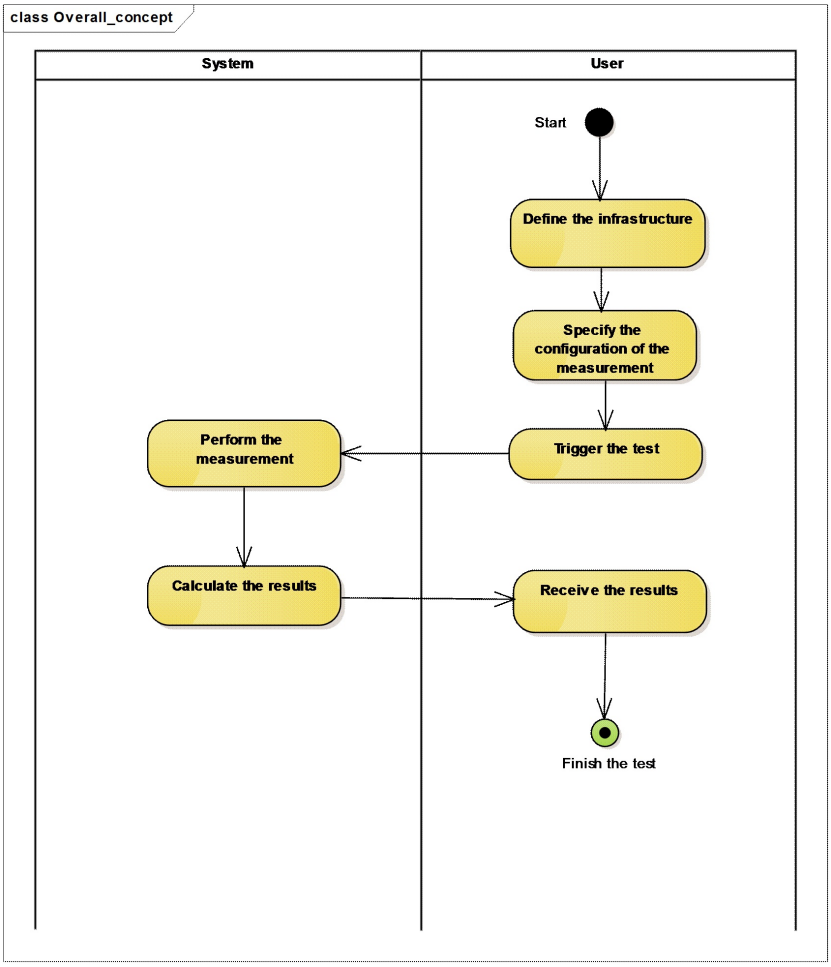


Figure 5.1: Workflow of the framework

5.5.2 Pre-measurement phase

5.5.2.1 Configuration file

Before performing the test itself, the configuration file has to be specified. Therefore the first task the program has got is to load and parse the configuration file in order to be able to process the steps specified there.

The following arguments are mandatory for the configuration file:

- ID

  The identifier of the test.

- Specification of tested VMs

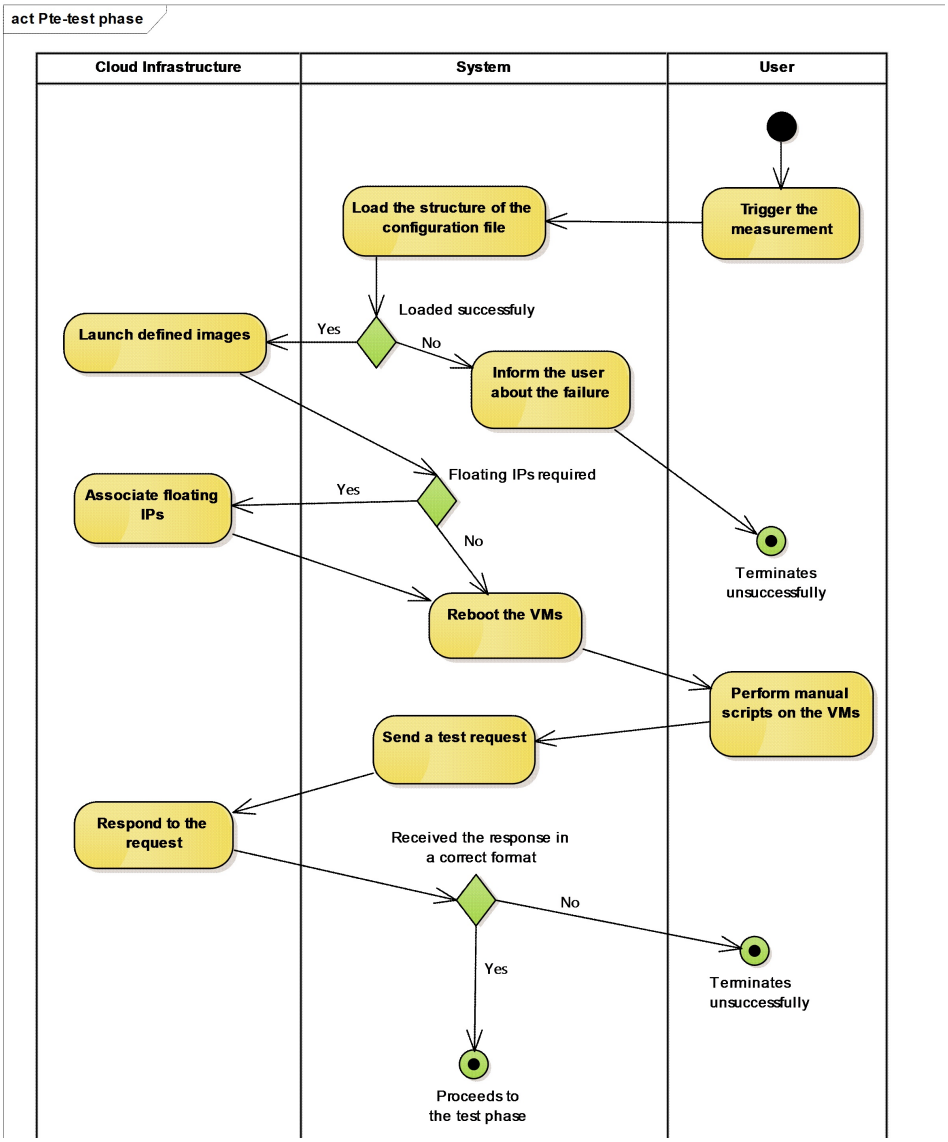


Figure 5.2: The pre-measurement phase

There are all the VMs specified here. Each server intended to be created must have a specification of the image to boot, the deployment and the flavour. The image to boot can be a plain OS or an image that was prepared in advance. Launching the VM can be associated with a script that is triggered on the VM after the boot.

Moreover, a request to associate a Floating IP to the VM can be

inserted.

- URL

  An URL where the availability can be tested. The URL is available when the system is operating and not available or responding incorrectly during the downtime. It has to be possible to distinguish the target nodes on the basis of the received requests so that each machine gives different response.

  When specifying the URL, it can either be a website or an URL on VM that receives a floating IP address.

- Output

  The filename of the output file where the results will be published to the administrator.

- Failover trigger

  A Bash command or a Bash script to simulate the downtime of a primary machine, thus to perform the failover.

- Credentials

  Credentials for the account from which to deploy the VMs. Such credentials usually include authentication URL, username, password, and tenant name or tenant ID. Nevertheless, the password is not required because it is better not to force the users to store it. The password is inserted manually when starting the measurement. Moreover, the authentication URL is composed automatically in accordance to the deployment[14] it is needed for. Therefore, it is not necessary as well.

  On the basis of the provided credentials, the VMs can be launched in desired deployment. The authentication URL is parsed and changed for the specific need of every deployment.

- Timeout

  The maximum time it is possible to take to provide the failover. If no completed failover within this time is detected, the measurement is terminated with a failure status.

It is however possible to specify other settings. These settings are optional with a default value if not specified.

[14]The term deployment describes a self-sufficient cluster of hardware components running the full installation of a cloud platform. The nodes of the same geographical location are included in the deployment.

- RPS

  Requests per second defines the maximal number of requests intended to be sent. If RPS is too low, the quality and precision of the measurement decreases. If RPS is too high, the target server might get overloaded and thus fail down. According to [9, p. 102-105], brute-force flood attacks are the simplest forms of service attacks and are the easiest to defend as well. Presuming the tested service features such a defense mechanism, it has to be taken in concern as well.

  If not specified, 300 seconds is the limit.

- Notes

  Notes and description of the test. It serves the administrator to describe the purpose and the parameters of the test.

- Incorrect responses

  The list of incorrect responses is specified here. If the list is defined, every response is checked whether it matches the items in this list. If so, the response is treated as incorrect. If not, the response is correct. This list enables to mark the responses that should not be taken in concern as responses from a healthy system. Thus, it helps to distinguish the availability and unavailability of the whole installation.

The configuration file is loaded and parsed. The mechanism can fail in several occasions:

1. The configuration file does not exist or is unable to be accessed.
2. The content does not meet the syntax of preferred language.
3. A mandatory item is missing.
4. The content does not fit the required format.

If parsing the configuration file ends unsuccessfully, the application is terminated with an error state. Only when the configuration file contains all necessary information and it was parsed successfully, the framework can begin deploying the defined system.

5.5.2.2 Deploying and configuring the infrastructure

After the configuration file is loaded, the program has enough information to launch VMs. Therefore the VMs can be launched as specified in the configuration file. It is possible to launch VMs from previously prepared images, to use a script to set it up or to combine these two ways.

Once the VMs are running, it is possible to attach a Floating IP address to the primary node if necessary so that the service can be addressed over the internet. If there are any scripts specified in the configuration file, they are triggered after attaching the Floating IP.

5.5.2.3 Testing the deployed infrastructure

When the set up of the VMs is complete, a test request is sent. The test request checks that the environment is deployed properly. The request should return a response fitting the correct format specified in the configuration file. If so, the measurement itself can begin. If not, the test ends with no measurement because the application to be tested was not deployed properly.

Before launching the test itself, a timeout has to be set so that the application can terminate properly even in case of unexpected behaviour.

5.5.3 Measurement phase

The test consists in sending requests repeatedly to specified URL. There are three possible situations resulting the sending:

1. The response is correct. That means that the system is up and running. The response came either from the primary resource while it was still running or from the redundant resource after completing the failover.

2. The response has an incorrect format. It indicates that we were able to get a response from the node we target but the node was unable to process it correctly. Either the node we address is still not fully operating or there is another node that is not operating. It is possible that the node we are sending the requests to, is sending requests to other nodes so even if we get a response, it can be based on a wrong behaviour of another node contained in the service that is not operating at the time.

3. The request ends with no response. That means that we are sending requests to a node that is not communicating over the network – either the node is down or there is a network interruption on the path to the node.

The requests begin to be sent to the specified URL. The results are being collected and analysed in order to define the scenario described in 5.3.2. As soon as enough data is collected, the temporal measurement can be triggered.

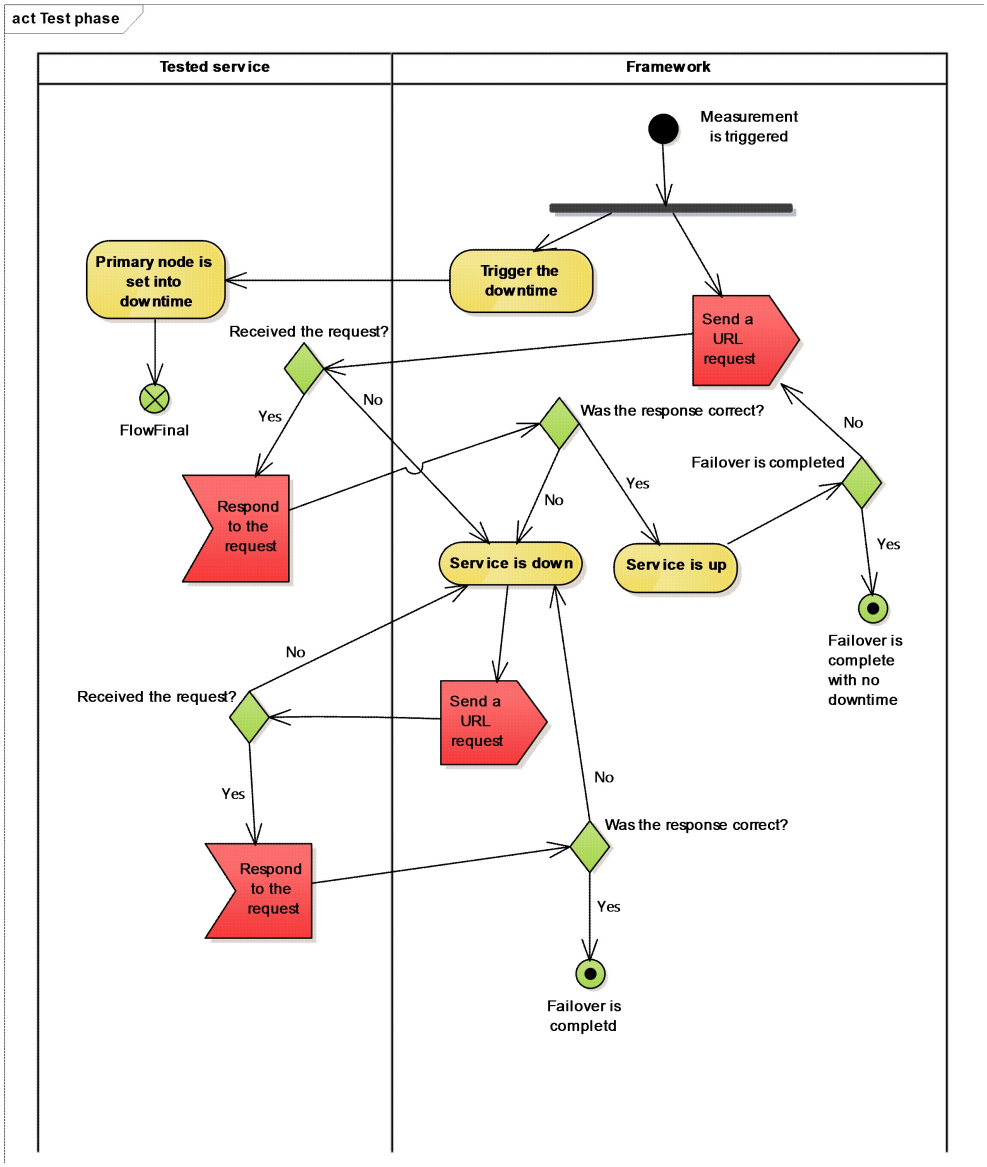


Figure 5.3: The measurement phase

The downtime is simulated as specified in the configuration file. That causes triggering the failover mechanism. In this period, the options 2. and 3. defined above may occur.

The behaviour during performing the measurement corresponds with algorithms and procedures defined in section 5.3.

5.5.3.1 Frequency of sending requests

The frequency of the requests sent is dependent on the device it is being sent from. It is possible to configure it up to 100 requests per second for every hardware setting. The more requests are sent the higher accuracy the measurement has.

When a measurement takes longer, the frequency of sending the requests is not needed to be so high. If a longer failover is being measured, the absolute precision does not have to be as high as during the shorter failovers. However, relative to the downtime length, the measurement still may have greater precision. The reduction of the frequency lowers the stress on both - the measured system and the measuring system. What is more, it also helps to reduce the amount of the data to store during and to evaluate after the runtime of the measurement.

5.5.4 Post-measurement phase

After the test is performed, all resources allocated to run the test case have to be deleted. This includes mainly VMs. VMs to delete correspond the VMs to create specified in the configuration file.

The measuring mechanism has collected data while performing the test. The evaluation of the measurement is performed using the data. The evaluation consists in defining the downtime length and the accuracy. Both downtime length and accuracy calculations are described in section 5.4

The calculated values are filled in an output file and distributed to the administrator. The target file is specified in the configuration file.

Chapter 6

Implementation

This chapter describes the technology and the processes used for the implementation. Moreover, eventualities are discussed to justify the selected technologies.

6.1 Technology selection

6.1.1 Configuration file

Because no large amount of data was expected, the main requirement for the data interchange format used in the configuration file was not the effectivity. The most important parameter was readability. On this basis, the main aim was to select an extensively used data format that most of the users understand. The configuration file was aimed to be easy to write and easy to understand.

JSON format was used as the best fit for satisfying the requirements. "*JSON is designed to be a data exchange language which is human readable and easy for computers to parse and use.*" [10]

6.1.1.1 Other eventualities

Other possible technologies include XML, YAML, and ConfigObj. However only XML was considered to be used because along with JSON, it is the most common data format.

Comparing XML to JSON as the two most common formats, JSON was chosen because of higher data/code ratio. That makes JSON faster to write and easier to read. On top of it all, according to [11], JSON proves to be a more effective format based on the performance test carried out there.

6.1.2 Programming language

The measurement of the failover is triggered as a Bash script that executes all the subtasks and shelters the whole process of the test. There are two main programming languages used for programming the framework. Python and C++. Usage of multiple programming languages is possible because the design of the solution sketches more separate phases. The phases are strictly divided from one another so no disruption of simultaneously running services is possible.

Because it is possible to divide the whole flow of the application, both programming languages can be used without causing any difficulties to the coding process or to the flow of the program. On the other hand, it is possible to choose the best fit for the different subtasks. As a result, the code uses the more advantageous language to handle every particular part.

6.1.2.1 The pre-measurement phase

Python is used to handle the first phase – the pre-measurement phase – and a minor part of the last phase – the post-measurement phase.

The overall goal of the pre-measurement phase is to build up the installation on which the measurement is performed during the subsequent phase. This task consists of loading the information stored in the configuration file and building up an infrastructure defined by the provided data.

The measurement is performed in the OpenStack infrastructure out of several reasons. Primarily, that is made because the infrastructure of Cloud & Heat runs on OpenStack infrastructure. OpenStack is an open-source platform enabling providing the infrastructure-as-a-service. It provides a RESTful API to enable the communication.

The native binding for OpenStack API is written in Python. Using the Python binding is a more transparent way of communicating with OpenStack cloud infrastructure than directly using the OpenStack API out of the developer's point of view. It is easier to code and to maintain as the functionality is already implemented in the python-openstack library. The Python binding is used to implement the command line interface as well. That leaves Python and Bash as a possibility for manipulating with the infrastructure. Because of author's preference, Python was chosen.

As JSON is used as a format of the data interchange, Python enables an easy parsing of JSON because it has a built-in library handling the import of the file. Therefore, it is easily possible to serialise it and thus to use the data defined there.

6.1.2.2 The measurement phase

The important parameter of choosing the programming language for the measurement phase is its time and memory efficiency and scalability. This is neces-

sary in order to provide a time-efficient algorithm of sending the URL requests to the target and processing the responses. The time efficiency is desirable in order to be able to send enough requests and process the received responses so that the measurement has enough precision. Because large amount of data during the measurement is possible to be collected, it is important to enable easy and effective processing of the data during the measurement.

As mentioned in the Design chapter, there are multiple threads used to handle the flow of the measurement. That presents the second condition important for choosing the programming language – an easy manipulation with POSIX threads and the availability of intercommunication between the threads itself. POSIX threads are available both on Linux and Mac OS X so it is a legitimate technology to use to fit both required operating system platforms.

The author chose C++ as the programming language of this process. It was chosen not only because it meets the requirements stated above. Another reason for choosing C++ was the fact that the author has a previous experience with several programming languages influenced by C, such as C++, C# and Java. Unlike C, C++, C# and Java are object-oriented languages. Moreover, C and C++ can be easily distributed as a source code and compiled on the target device. The source code is compiled into executable machine code. On the other hand, Java and C# need an execution runtime and it don't enable so deep scalability and are not as efficient because of a higher level of abstraction that is used.

6.1.2.3 The post-measurement phase

The data regarding the measurement is gathered while running the C++ code and thus is stored in form of variables that point to addresses the data is in. As defined in sections 5.3 and 5.4 the analysis of the data is performed as a part of the measuring logic. Thus, the values regarding the failover are calculated as a part of the testing code.

The infrastructure for the measurement was built in Python using the python-openstack API. The deletion has the same demands as the building so the same process is applied.

6.1.3 Scripts launched on the VMs

There are scripts performed on the VMs after the boot before the assignment of the Floating IP address. If such scripts are specified for the particular VM, the cloud-init service is used. The cloud-init is a service that is performed on OpenStack VMs during the boot-time. It is performed with super-user rights. An important functionality of the cloud-init service is importing SSH

keypairs[12, p. 76]. Another advantage is its capability of running a userdata script in the interpreter specified on the line with sha-bang[15].

After executing this set of scripts, a Floating IP address is associated to the VM if specified. After this moment, it is still possible to run scripts manually on the VMs, for example via the SSH connection. This possibility should serve to enable executing commands on VMs with already associated IPs. Therefore, the public IP address can be taken in consideration for the scripts that are being performed. The opportunity to run the scripts manually should also serve for finishing the configuration manually if scripting it is too complicated. From this point on, the installation is considered completed and the failover mechanism is running.

6.1.4 Included external libraries

The framework uses several libraries in order to handle common processes such as sending URL requests or parse the configuration file.

- jsoncpp

  Jsoncpp[14] is a library used for serialising and de-serialising the configuration file in the C++ code.

- commentjson

  "*Commentjson is a Python library that lets you have Python and JavaScript style inline comments in your JSON files. Its API is very similar to the Python Standard library's json module.*" [15] The possibility of using comments in JSON is not enabled. Commentjson library enables easier writing of configuration files.

- libcurl

  Libcurl [16] is a library handling transferring URL requests supporting various protocols. It is a C and C API parallel to command line tool cURL.

- pycurl

  PycURL[17] provides an interface for libcurl for Python.

- python-novaclient

  Python-novaclient[18] is a python client for communicating with OpenStack Compute nodes. It provides the binding to the OpenStack Nova API.

[15] "*The sha-bang (#!) at the head of a script tells your system that this file is a set of commands to be fed to the command interpreter indicated*"[13, p. 4]

6.2 Processes in the framework

This section contains a description and justification of several processes implemented in the solution.

6.2.1 Building the infrastructure

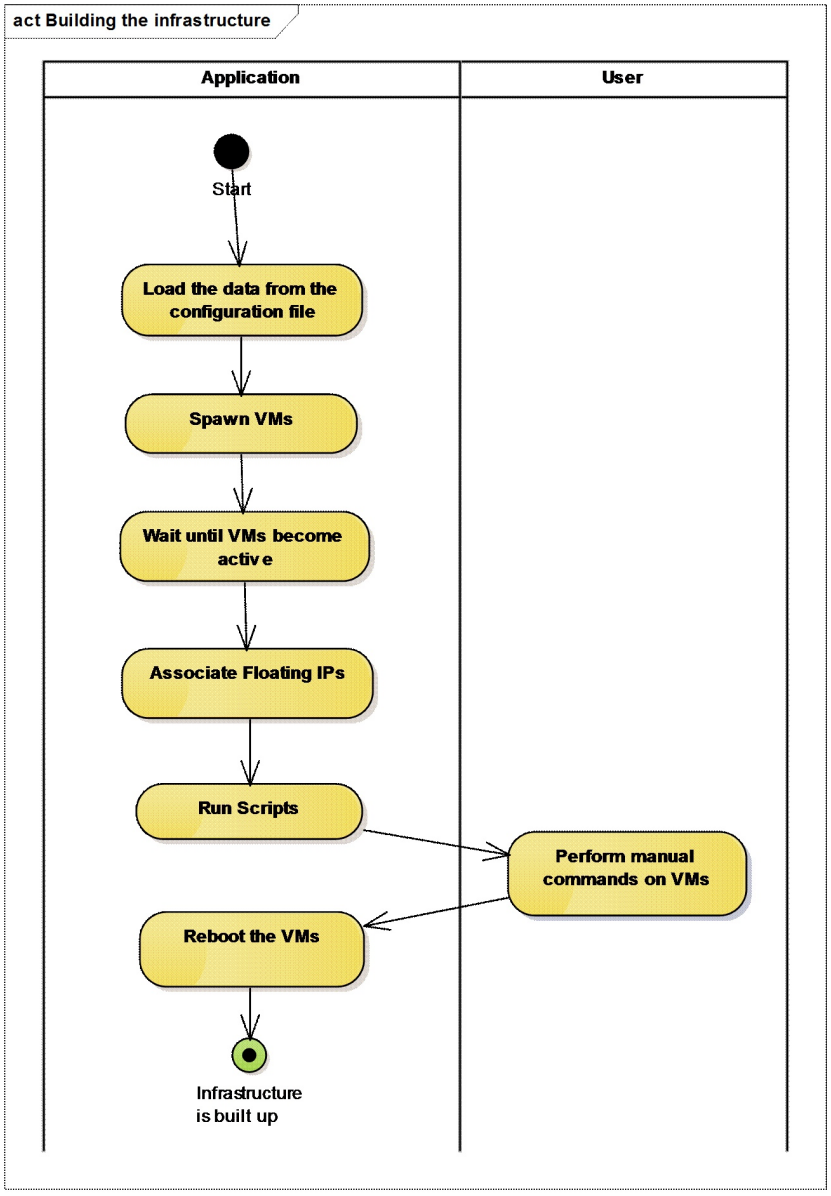


Figure 6.1: Building the infrastructure

Building the infrastructure for the measurement is the main part of the pre-measurement phase. The infrastructure is built in order to provide the highly available service for performing the measurement. Because setting up the highly available installation might be a complex process, a procedure enabling an easy use of the framework was designed.

As mentioned in section 5.5, the configuration file spawns VMs from images in order to enable testing a VM that is the closest to the real one deployed in the production environment. It is not a complicated task to create a snapshot of an instance in a cloud environment. Still, the snapshot itself does not solve network bindings used to communicate in the network. A new VM receives a random private IP from the private IP pool of the deployment it is spawned in. Moreover, when associating a Floating IP in order to enable the VM to be addressed over the internet, the Floating IP is gained from a Floating IP pool. It is not always possible to choose the particular IP address to associate with the VM. Beside many other possible aspects, this is the reason, why performing scripts on VMs after it is spawned is enabled.

There are two moments during the whole phase enabling to run a script on a VM in order to enable configuring the high availability. Such situations are described and justified in subsection 6.1.3. Moreover, the tested infrastructure requires an invariable content able to be retrieved on a URL specified in the configuration file. For the sake of performing the measurement itself, it is desirable for the VMs to respond with a message in different format. Only this setting enables correct determination of the precision of the measurement.

In order to simplify and shorten the preparation for the test, a possibility to perform manual scripts on VMs was added as well.

The activities are performed one after another to ensure building the infrastructure in the right order. This set of conditions defined the flow as described in the activity diagram 6.1.

6.2.2 Sending the requests and receiving responses

During the whole runtime of the measurement, the requests are sent to the target service in order to find out whether the service is operating at a time. As mentioned in section 5.3, the mechanism that handles sending the requests runs parallel to the mechanism processing and evaluating the results.

For the sake of the quality of the measurement, it is important that the frequency of sending the requests is invariable to satisfy the preferred RPS[16]. The requests must not be sent too fast one after another so that the target server manages to respond to the requests and no security mechanism blocks the incoming IP address. On the other hand, if RPS is too low, the measurement loses its precision.

[16]Requests Per Second

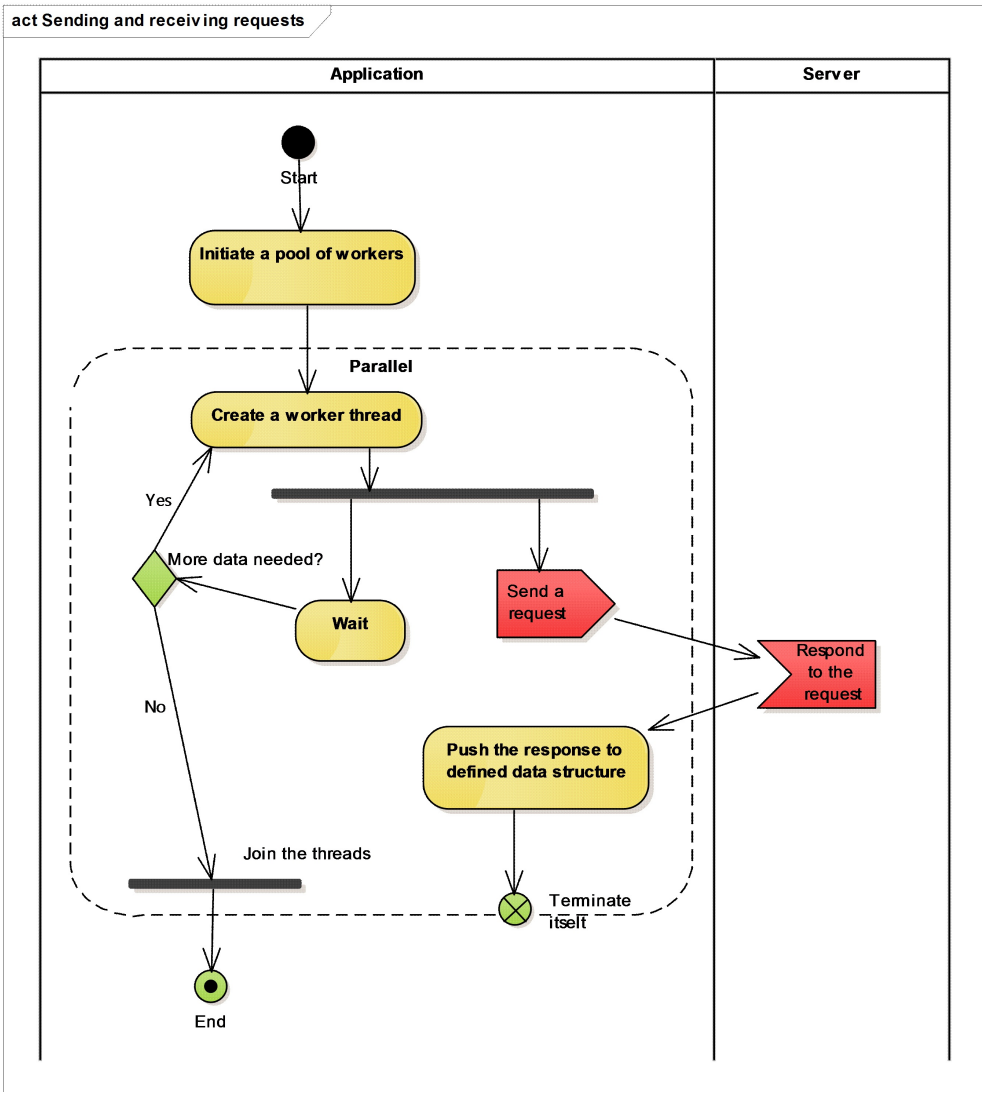


Figure 6.2: Sending the requests and receiving the responses

To fulfil this requirement, there are more threads needed for handling the sending and the receiving of the requests. Considering evaluating the results from an available and unavailable service, the time needed to handle sending and receiving the request varies significantly. When a machine is fully operating, a request normally takes roughly one hundredth to one tenth of second. If a service is unavailable, it may take much longer until the request times out. The length is dependent on the parameters of the request. Anyway, the request can take milliseconds or seconds to be evaluated.

That determines the need of sending the requests in parallel design not waiting for the preceding requests to be completed. The used design creates

one thread in time intervals defined by the RPS. Every worker thread performs one request and terminates.

Another possible approach is to spawn multiple worker threads handling request by request in sequence as an analogy to the producer-consumer problem. Although it features better performance results by not having to create a thread for every request, in the result it stresses the system more. It is caused by the fact that all threads access the mechanism ensuring the maximal RPS. With a mutex lock and a sleep sequences to reduce the impact, the entropy of RPS is increased while the system still must handle heavier load. Moreover, such a solution is more hardware-sensible.

6.2.3 Processing the received data

The responses to the requests are processed in different ways in different phases of the measurement. The aim was to design an algorithm that uses the least possible memory because the amount of received data is not limited.

6.2.3.1 Collecting statistical data regarding the usage of the VMs

Before running the command to trigger the failover mechanism, the behaviour of the fully available system has to be discovered. The procedure is described in subsection 5.3.2.

In order to collect the data, the results are filtered according to the responses that represent the different VMs that have handled the requests. There is a list of VMs with its request times kept in the memory. On the basis of this data, the number of primary VMs is discovered.

What's more, the number of responses necessary to keep is calculated. It is possible that the measurement takes minutes and thus a lot of data is received. If all of the data were stored, the system would soon run out of memory. The move against such a threat was to free all unnecessary data immediately so that the amount of the stored data is not proportional to the length of the measurement but is constant.

6.2.3.2 Evaluation of the failover

After the failover mechanism is triggered, there is only a limited amount of responses to requests kept in the infrastructure. The necessary amount is set on the basis of the data collected as described in the previous subsection.

When a new response is received, it is pushed to the list with responses, the oldest response is popped out of it and its resources are freed. Such a behaviour ensures not more that necessary memory allocated.

When a response is received, it is analysed and calculations described in 5.3.2 are performed in order to determine the current status of the tested infrastructure.

6.2.4 Multi-thread processes

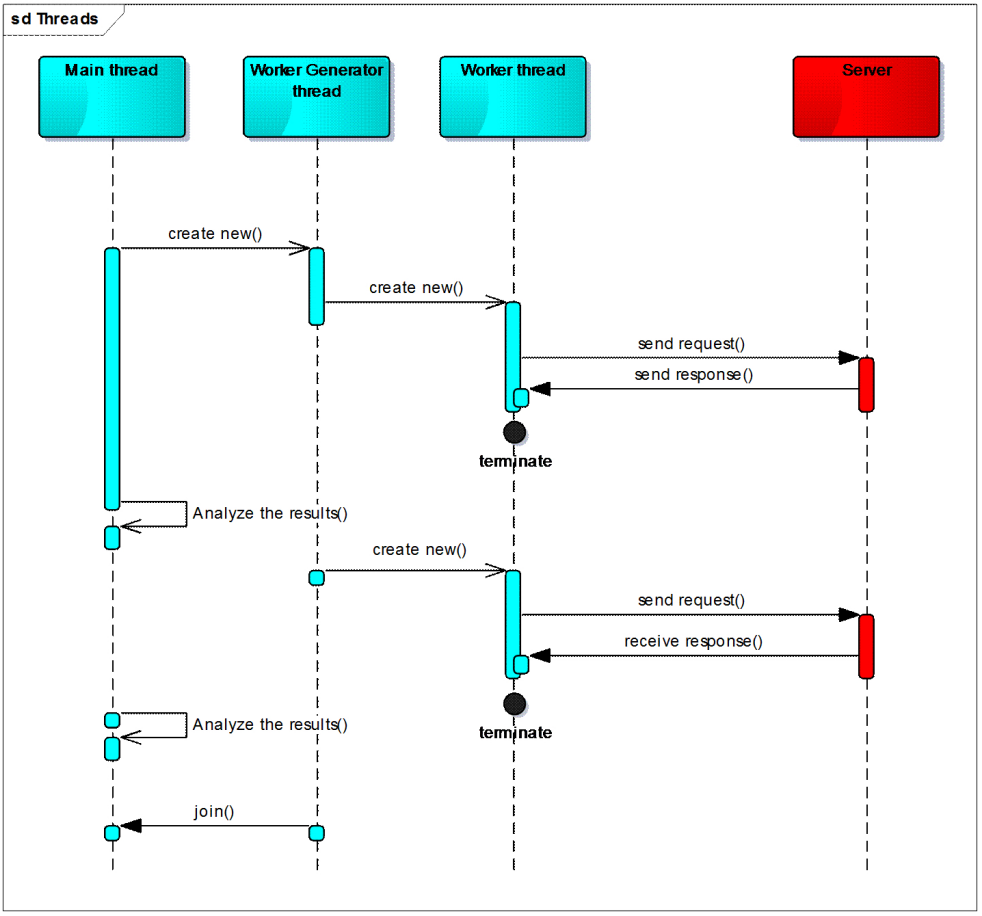


Figure 6.3: Threads

There were several algorithms processed in separate processes described earlier in this section. The measurement phase operates in three main lines:

1. The main thread

   This thread handles mainly the analysis of the responses.

2. Generator thread

   This thread is responsible for launching worker threads and servicing the resources of worker threads.

3. Pool of workers

   Every thread from the pool sends the URL request, receives the response, and terminates.

The relation of the threads is visualised in 6.3. The generator thread is created by the main thread. Then it controls launching worker threads in given time intervals.

The intercommunication of the threads is solved with help of shared resources. The threads are in hold of pointers to structures and variables it administrates. That indicates that there are certain structures that are accessed from multiple threads. When more threads administer a single resource, data integrity has to be ensured so that it is not possible to manipulate with a single resource from multiple threads at a time. There were two resources identified as critical.

1. The list of the responses.
2. The flag determining the availability of a worker thread.

Securing of such a scenario where two threads want to access or modify a single variable is provided with mutex locks. It ensures that only one thread accesses the critical part of the code and thus the critical resource at a time. The code locked with mutex is as short as possible. In other case the performance is affected.

6.3 Output file structure

The output file contains the results of the measurement. There is no need for choosing another file format for the output file structure so its format is JSON as well. There are the same requirements for the output file as for the configuration file – the file is easy to read and to parse. On the other hand, as well as the configuration file, the output file does not contain too much items in it and therefore the speed of its processing is not an important criterion.

The output file contains the following fields:

- ID

  ID of the test as specified in the configuration file.

- Timestamp

  Date and time of the measurement.

- Downtime duration

  The measured value of the downtime duration.

- Precision

  Precision of the result.

- Infrastructure

  Specification of the infrastructure taken from the configuration file.

- Notes

  Notes taken from the configuration file.

6.4 Summary

The framework was implemented on the basis of the design of the solution provided in chapter 5 taking available technologies in concern. The implementation targets the needs of the framework.

It aims at the source code to be easily-sustainable. That is achieved by creating a fine-grained architecture. This includes using open-source third-party libraries and framework that help solve common tasks. It is possible to update only a certain part of the application without influencing the other parts in such an infrastructure. That makes the code easy to maintain and update.

The choice of the preferred technologies was made in accordance to the main needs of the framework, author's previous experience with the technologies, and its popularity across the community.

Chapter 7

Evaluation

7.1 Technique of the evaluation

The framework was evaluated in order to ensure that it is working properly.

Due to the main functionality of the system – the ability of measuring the temporal behaviour of the failover – there was no possibility of creating a set of tests and comparing it with a set of correct results. Because of the fact that the duration of a process is being measured, the results were expected to deviate even with the same input data. There are more aspects causing the deviation:

1. The testing takes place over the internet.

   The requests are being sent over the internet as well as the VMs communicate with one another mostly over the internet. The communication over the internet is not invariable when considering its temporal characteristics.

2. The infrastructure is most likely built on different hardware.

   The VMs are spawned on hardware components unable to be defined by the user himself. The hardware is allocated automatically, the user can only set the deployment to spawn the VM in.

   When having the underlying layers not under control, the VM, although always with similar qualities, do not always have to run on the identical type and even piece of hardware. This might cause slight performance deviations that might affect the results of the measurement.

3. The downtime command hits the system in different state.

   For example, heartbeat is a popular method of gaining the information of the health status of a component. It consists in sending a

signal to the scanned component in given intervals. By responding to those signals, the component states its status.

Since the heartbeat is sent in intervals, it is not sure in which moment referred to the last heartbeat signal the downtime command is performed. For example if a machine is set to downtime right after it has responded to a heartbeat signal, the instance that sends the heartbeat would consider it operable the whole period until a new heartbeat is sent.

This set of conditions prevented the author from writing unit tests to test the system automatically. Beside the factors causing the deviation, it was made difficult because of the lack of the tenant-independency. That is caused by the fact that every user runs his own tests on the infrastructure defined himself. It is of course not possible to access somebody else's infrastructure and thus to launch VMs from snapshots created by this person. That means that only the person (tenant) that have defined the infrastructure is able to run tests using that infrastructure.

Another aspect making the automated testing difficult was the absence of any reference data. That means that it was not possible to declare that the programme is working after appropriate results are achieved. The only tool helping to verify the temporal results achieved in the measurement in at least an approximate way, were scripts written by the author himself. Such scripts were however far from being as precise as the framework itself.

7.1.1 Areas of focus

As it was not able to automate the testing process, the framework was tested manually. The main areas of focus when evaluating the results were the following:

- Incorrect input.

  The application was tested with incorrect input in order to see that it does not fail in an uncontrolled way. The main areas of the evaluation were testing the behaviour after setting an invalid configuration file, wrong credentials and unfitting infrastructure.

- Limits of the infrastructure.

  The behaviour was examined when infrastructure limits are being exceeded. That includes attempts to spawn more VMs than allowed for a tenant in a deployment or attempts to allocate more floating IPs than allowed.

- Different functionality.

All different kinds of configuration were examined in order to test the functionality of every part of the possible configuration.

- Unsuitable architecture of the failover approach.

  An unsuitable architecture was deployed. That includes configuration that spawns a service that is not available on specified URL or does not provide failover after all.

- Excessively fast and slow approaches.

  Abnormally time effective approaches were tested to assure the right behaviour even with less or no downtime responses received.

- Normal data.

  Normal data expected to be used were tested as well.

7.2 Fulfilment of the requirements

The implemented software was evaluated as described in the previous section. On the basis of the evaluation, the fulfilment of the requirements was discussed.

The functional and non-functional requirements are the conditions that define the system. It must fulfilled in order to be able to achieve the main task. The requirements were specified on the basis of a use case that addresses a typical use of the framework.

F1 Measure the duration of a downtime.

The requirement is fulfilled. The measured downtime length is a part of the output file.

F2 Store the information regarding the physical location of all interacting instances.

The requirement is fulfilled. All VMs launched in order to test the selected failover approach are displayed in the output file. The deployment where the VM was launched is a part of this information.

F3 Store the information regarding the accuracy of the measurement.

This requirement is fulfilled. The precision of the measurement is provided as a part of the output file.

F4 Provide the results in a unified format.

The requirement is fulfilled. The results are provided in a unified JSON format. The file with results always contains the same set of information. The values are stored under the same keys. Thus, the structure is invariable and only the values change.

F5 Run the measurement for different configurations of failover approaches.

The requirement is fulfilled. It was tested with both active/active and active/passive approaches. Still, it is not assured that all active/active and active/passive failovers are able to be measured. In order to assure it, all failover approaches would have to be tested.

F6 Do not affect other services running in the same infrastructure.

The requirement not entirely fulfilled. The framework allocates only unused resources and thus does not influence other services running in the infrastructure. On the other hand, the framework launches VMs and triggers scripts if some are defined. The workload of the launched VM or the content of the script may affect other services because the actions being made are not being supervised. Nevertheless, it is not expected that the configuration would be made to affect other services.

F7 Release all allocated resources after the test.

The requirement is not entirely fulfilled. All resources specified in the configuration file are deleted after the test is performed. Similar to requirement F6, it is not impossible to allocate further resources as a part of the workload that is performed on the VMs. Thus, these resources may remain in the infrastructure after the test is finished if the VM does not assure its deletion on its own.

F8 Store the configuration of the measurement.

The requirement is fulfilled. The configuration of the infrastructure that was built up in order to provide the measurement is saved in the output file as well as the results of the temporal measurement.

N1 Runs on a typical cloud platform.

The requirement is fulfilled. The framework was tested in the infrastructure of Cloud & Heat. Such an infrastructure runs OpenStack.

N2 Communicates over the internet.

The requirement is fulfilled. It is enabled that the VMs can communicate over the internet. This is enabled by the possibility of

associating Floating IP addresses. The floating IP address is public, therefore it is possible to address the VM directly over the internet.

N3 Can be launched from Linux distributions and from Mac OS X.

This requirement is fulfilled. The framework was tested both on Linux Ubuntu 14.04 and Mac OS X El Capitan 10.11.4.

N4 The test is performed with a low margin of error.

This requirement is fulfilled. The result of the measurement is provided with a milliseconds accuracy. Nevertheless, in order to achieve the milliseconds precision, the tested infrastructure has to be able to bear a high frequency of sending the requests. Moreover, the internet connection is a factor affecting the precision as well. On the other hand, the expected maximal precision to be tested is expected to be tens of milliseconds. When such a precision is required, it is achieved.

In conclusion, the majority of the requirements is fulfilled. It enables using the implemented test framework in the way it was designed.

The requirements F6 and F7 are not entirely fulfilled because the framework has no control over the processes in the infrastructure it has built up. That is however not necessary because it is not a common use case where the described behaviour appears. Moreover, it is not threatening the measurement itself and according to the design of the framework, it is only possible to affect the own tenant. That means that the framework does not enable the user to do more than he is able to do from the command line. In other words, it is not able that the framework gets abused to access or modify the infrastructure that the user is normally unable to access.

7.3 Sample measurements

As the evaluation focussed on correct behaviour of the framework, measurements were performed on the application. The measurements focussed on common solutions that are being used in order to increase the availability of cloud services.

7.3.1 Nginx load balancer

This scenario is deploying three VMs in three deployments. There is one instance running an Nginx reverse proxy server and two upstream nodes running an Apache web-server. Because the servers are located in different deployments and thus are not in the same private network, public Floating IPs were used.

The Nginx proxy server configuration put equal stress on both upstream nodes, thus every node serves a half of the requests when the system is operating.

The installation that was deployed was providing an active/active failover.

The measurement was performed with 100 RPS. After five runs, the result was always a zero downtime. The average precision was 21 ms and standard deviation 1 ms.

The result indicates that there was either no downtime of the service at all or that the downtime was less than the precision of the measurement (21 ms) so it was not captured.

The Nginx load balancer seems to be a very effective solution in this configuration. When the service is stateless, load balancing is an easy and effective way how to increase the availability of a service.

7.3.2 Nginx with primary and backup node

This scenario deploys the same infrastructure as in the previous case. There is however a different configuration of the load balancer on the proxy server node. The configuration defines one primary upstream server and one backup server. That means that when the system is fully operating, all the requests get redirected to the primary server. When the primary server fails, the backup server takes over. After the failover, all requests are handled by the backup node as long as the primary server is unavailable.

The installation that was deployed was providing an active/active failover. Although the second node is stored as a backup node, it is online for the whole time so it is active as well.

The measurement was performed with 20 RPS. The tests performed on the same infrastructure as in the previous case present totally different values. The average downtime length of 7 tests that were performed is 97952 ms with a standard deviation 32596 ms. The average precision was 51 ms and standard deviation 1 ms.

Because of the high standard deviation, the result is not reliable. Actually, none of the particular results was close to the average value. The results were either close to 60000 ms or to 120000 ms. This might be possibly caused by some mechanism on the load balancer node running with a period of 60000 ms.

On the basis of the performed tests and its results, it can be inferred that such a solution is not a suitable choice when designing a highly available installation and it is better to avoid it if possible. This approach have a very bad performance values and have proven to be unreliable. Other explanation of the unsatisfactory results might be a wrong configuration of the load balancer node.

7.3.3 Reassigning a Floating IP with Corosync and Pacemaker

This scenario deploys two VMs in one deployment. The failover is provided by reassociating the Floating IP address. The Floating IP is always associated to one of the VMs. The VMs run in cluster with Corosync and Pacemaker as a failover solution. The cluster reassociates the Floating IP address in case of a downtime. The tested installation features an active/passive failover.

The failover approach is customised to the OpenStack infrastructure. It features an ad-hoc resource agent managing the Floating IP address.

The measurement was performed with 50 RPS. There were 5 measurements performed on this infrastructure. The average downtime was 5925 ms and standard deviation 912 ms. The average precision was 21 ms and the standard deviation 1 ms.

The performance of this solution is not as high as in the first test. On the other hand, the solution features an active/passive failover that is less efficient by default. Still, the measured downtime is short and the solution is reliable according to the low standard deviation of the downtime length. When load balancing is not possible, Corosync and Pacemaker might be another reliable and relatively fast failover approach.

However, because a custom resource agent is used, the measured values are not universal. Another installation with Corosync and Pacemaker using other resource manager might have different temporal behaviour during failover.

After running and evaluating several tests, it is clear that the temporal effectivity of different failover approaches varies a lot. It has been shown that even a minor change in the configuration of the failover mechanism can cause big differences.

Conclusion and Outlook

The goal of the theoretical part of the thesis was to design a test framework to measure temporal characteristics of failover processes in cloud environment. The subtasks specified in the description of the topic were met one after another and lead the author towards the final shape of the solution.

At first, the state-of-the-art analysis of existing failover approaches was provided in chapter two along with the terminology used in the high availability field.

On the basis of the requirement analysis, the design of the solution was made reflecting the analysis of the used approaches.

The practical part consisted of implementing the framework according to the design and the implementation details specified in the theoretical part. The solution was implemented to fit the infrastructure of Cloud & Heat Technologies GmbH.

Finally, the implemented framework was tested and the results of the tests were discussed.

In conclusion, the goals of the bachelor thesis were accomplished.

The possible future development of the framework includes implementing provider independency. It is possible to modify the framework in order to be applicable to all OpenStack vendors with the price of more extensive configuration in order to specify the vendor's endpoints. Moreover, the framework could get partially platform independent. For example, the OpenStack API is compatible with Amazon EC2 API.

In the future, the framework could serve as a tool for doing a comprehensive analysis of failover approaches in cloud environment. The results of the analysis would be useful for having an overview of the cloud failover approaches' temporal qualities. Nevertheless, the most reliable method for predicting the temporal behaviour of a failover approach is to test it on the same infrastructure as it would be used in.

High availability is an important aspect of almost every solution running in the cloud. Still, it is a complex problematic offering a lot of different solutions with different qualities. Hopefully this work can help to select the best fit and evaluate its qualities.

Bibliography

[1] OpenStack. OpenStack Docs: High availability concepts. 2016, [Online; Accessed 2016-03-13]. Available from: `http://docs.openstack.org/ha-guide/intro-ha-concepts.html`

[2] Schmidt, K.; Books24x7, I. *High Availability and Disaster Recovery: Concepts, Design, Implementation*. DE: Springer-Verlag, first edition, 2006, ISBN 9783540244608; 3540244603; 9783642063794; 3642063799; 9783540345824; 3540345825;.

[3] ClusterLabs. 2016, [Online; Accessed 2016-03-13]. Available from: `http://clusterlabs.org`

[4] Resource Agents - Linux-HA. 2016, [Online; Accessed 2016-03-13]. Available from: `http://www.linux-ha.org/wiki/Resource_Agents`

[5] Hoff, T. ZooKeeper - A Reliable, Scalable Distributed Coordination System. 2016, [Online; Accessed 2016-03-25]. Available from: `http://highscalability.com/zookeeper-reliable-scalable-distributed-coordination-system`

[6] Aivaliotis, D. *Mastering Nginx*. Packt Publishing Ltd, 2013, ISBN 1849517444.

[7] Gray, J.; Siewiorek, D. P. High-availability computer systems. *Computer*, volume 24, no. 9, 1991: pp. 39–48.

[8] Li, Z.; Liang, M.; O'brien, L.; et al. The Cloud's Cloudy Moment: A Systematic Survey of Public Cloud Service Outage. *International Journal of Cloud Computing and Services Science*, volume 2, no. 5, 2013: pp. 321–330.

[9] Ristic, I. *Apache security: the complete guide to securing your Apache web server*. Sebastopol: O'Reilly, first edition, 2005, ISBN 0596007248;9780596007249;.

[10] Nurseitov, N.; Paulson, M.; Reynolds, R.; et al. Comparison of JSON and XML Data Interchange Formats: A Case Study. *Caine*, volume 2009, 2009: pp. 157–162.

[11] Simec, A.; Maglicic, M. Comparison of JSON and XML Data Formats. Faculty of Organization and Informatics Varazdin, 2014, pp. 272–275. Available from: http://ezproxy.techlib.cz/login?url=http://search.proquest.com/docview/1629618564?accountid=119841

[12] Jackson, K.; Bunch, C. *OpenStack Cloud Computing Cookbook*. Community experience distilled, Packt Publishing, 2013, ISBN 9781782167587. Available from: https://books.google.cz/books?id=LxkrngEACAAJ

[13] Cooper, M. *Advanced bash-scripting guide*. lulu.com, 2010, ISBN 978-1435752184.

[14] open-source-parsers/jsoncpp: A C++ library for interacting with JSON. https://pypi.python.org/pypi/python-novaclient, [Online; Accessed 2016-04-30].

[15] commentjson - Add comments in JSON files. https://commentjson.readthedocs.org, [Online; Accessed 2016-05-13].

[16] libcurl - the multiprotocol file transfer library. https://curl.haxx.se/libcurl/, [Online; Accessed 2016-04-30].

[17] PycURL Home Page. http://pycurl.io, [Online; Accessed 2016-04-30].

[18] python-novaclient 4.0.0. https://github.com/open-source-parsers/jsoncpp, [Online; Accessed 2016-04-30].

Acronyms

API Application Programming Interface

Bash Bourne again shell

DB Database

CRUD Create, Read, Update, Delete

CUD Create, Update, Delete

DNS Domain Name Server

DRS Distributed Resource Scheduler

HA High availability, Highly available

IaaS Infrastructure as a Service

JSON JavaScript Object Notation

LAN Local Area Network

LSB Linux Standard Base

OCF Open Cluster Framework

OS Operating System

PaaS Platform as a Service

RPS Requests Per Second

REST Representational State Transfer

SaaS Software as a Service

SPOF Single Point Of Failure

SSH Secure Shell

STONITH Shoot The Other Node In The Head

URL Uniform Resource Locator

VM Virtual Machine

VPN Virtual Private Network

XML Extensible markup language

YAML YAML Ain't Markup Language

Contents of enclosed CD

- `readme.md` ..... the file with CD contents description
- `framework` ..... the directory of the framework
  - `config_files` ..... configuration files directory
  - `source` ..... source code directory
  - `run.sh` ..... run script
  - `readme.md` ..... readme of the framework
- `thesis` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `src` ..... the source code of the thesis